Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXI, págs. 3075-3084 (artículo completo)
Alberto Cardona, Paul H. Kohan, Ricardo D. Quinteros, Mario A. Storti (Eds.)
Salta, Argentina, 13-16 Noviembre 2012

GENERACIÓN DE MALLAS DE TETRAEDROS DELAUNAY EN PARALELO A PARTIR DE UNA NUBE DE PUNTOS Y UNA FRONTERA IMPUESTA

Pablo J. Novara y Nestor A. Calvo

Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral, Ciudad Universitaria, Santa Fe, Argentina, pnovara@unl.edu.ar

Centro Internacional de Métodos Computacionales en Ingeniería - CIMEC (INTEC). Parque Tecnológico Litoral Centro, Santa Fe, Argentina.

Palabras Clave: Mallas, Delaunay 3D, tetraedrización, paralelización.

Resumen. En este trabajo se exploran algunos posibles caminos para utilizar el algoritmo de DeWall para generar mallas 3D Delaunay utilizando una arquitectura de memoria compartida. El algoritmo de DeWall propone una estrategia de tipo divide and conquer para distribuir el problema en diferentes hilos o procesos. En cada paso se construye una pared de elementos que separa el problema en 2 subproblemas completamente independientes, que pueden ser resueltos individualmente para luego unir los resultados de forma directa sin realizar ninguna modificación a las mallas parciales. En este trabajo se analiza la influencia de algunas estructuras de ordenamiento espacial sobre los tiempos de mallado, se discuten los problemas relacionados a la precisión numérica y la imposición de una frontera presentando las soluciones implementadas, y se plantean posibles mejoras al proceso general para aumentar la eficiencia paralela en los primeros pasos del algoritmo, ya que son estos pasos los más costosos del proceso y son además los pasos en los cuales habrá procesadores sin tareas asignadas. Finalmente, se discute la posibilidad de obtener beneficios utilizando el algoritmo en una arquitectura de memoria local.

1. INTRODUCCIÓN

Este trabajo se presenta como una continuación de Novara y Calvo (2011), donde se discutía la implementación de un método para generar la triangulación Delaunay en paralelo a partir de una nube de puntos. En este trabajo se resume rápidamente la metodología utilizada en la mencionada implementación 2D, se presentan las modificaciones necesarias para que el proceso respete una frontera impuesta (el método original en el cual se basa ese trabajo triangulaba el convex-hull del conjunto de puntos), y se discuten las dificultades que aparecen al extender el método a 3D, proponiendo algunas posibles soluciones.

El método en el cual se basan tanto la implementación 2D como la extensión a 3D (Cignoni et al., 1998), presenta algunas ventajas frente a otros métodos de paralelización de la generación de mallas a partir de una nube de puntos impuesta. En primer lugar, es un método de tipo divide a conquer, pero completamente desacoplado. Esto quiere decir que no requiere de una etapa de merging, sino que luego de subdividir el dominio en subdominios a repartir entre los distintos procesadores y resolverlos individualmente, los elementos resultantes conforman directamente la malla final, sin necesidad de adecuar las interfaces comunes. En un método acoplado, la etapa de merging puede ser costosa, requerir de mucha comunicación, o a veces estar basada en ordenamientos que solo son posibles en 2D. Por otra parte, los métodos desacoplados suelen utilizar para mallar cada subdominio algoritmos seriales convencionales, por lo que deben garantizar una correcta división de la geometría de forma que la carga resulte adecuadamente balanceada. El método utilizado se puede aplica recursivamente en forma serial para resolver los subdominios. Esto hace que la carga pueda rebalancearse sobre la marcha, sin necesidad de aplicar complejos algoritmos de particionamiento en la etapa de subdivisión. Es decir, se pueden utilizar subdivisiones triviales (ejemplo: planos medios), y corregir más adelante el desbalance de carga que pueda producirse. Cada subdominio generará nuevos subdominios que serán resueltos por el mismo procesador que los generó si la carga se mantiene correctamente balanceada, o distribuidos entre los procesadores ociosos en caso contrario. Además, la simpleza de la subdivisión inicial permite reducir la necesidad de comunicación en una potencial implementación en arquitecturas de memoria local, dado que si se utiliza por ejemplo un plano medio, basta con comunicar la ecuación del plano para que cada procesador pueda reconstruir la subdivisión. Por otro lado, también permite considerar estrategias para reutilizar una subdivisión existente (realizada por ejemplo por un solver de elementos finitos u otro proceso que utilice el mallador) como subdivisión inicial.

2. TRIANGULACIÓN EN PARALELO BASADA EN DEWALL

El método de referencia para triangular el convex-hull de un conjunto de puntos en el plano, genera primero una pared de triángulos definitivos que cumplen la condición Delaunay y dividen el espacio en dos regiones inconexas, y luego distribuye los subespacios resultantes en dos procesadores. De esta forma, la pared de triángulos permanecerá fija y las triangulaciones parciales generadas en paralelo se incorporan a la malla final sin modificaciones. Si la condición Delaunay se cumple en todo el dominio, y se tratan adecuadamente los casos de ambigüedad, se puede garantizar que las mallas parciales generadas en un subdominio se acoplan perfectamente a los fragmentos de mallas ya generados en las paredes que lo delimitan. El algoritmo en paralelo puede utilizar un pila o cola de trabajos, donde cada trabajo consiste en la construcción de los elementos que conforman la pared que separa un dominio en dos nuevos subdominios, y debe ser realizado por completo en un único procesador. En cualquier punto del algoritmo, todos los trabajos presentes en la pila o cola de trabajos son independientes entre sí. Luego de

realizado el primer trabajo, ya se tienen dos trabajos que pueden resolverse en paralelo. Y luego de resueltos estos dos, cuatro, y así sucesivamente, hasta que los dominios de los trabajos sean lo suficientemente pequeños como para no generar nuevos subtrabajos (por ejemplo, cuando el dominio solo contiene tres nodos).

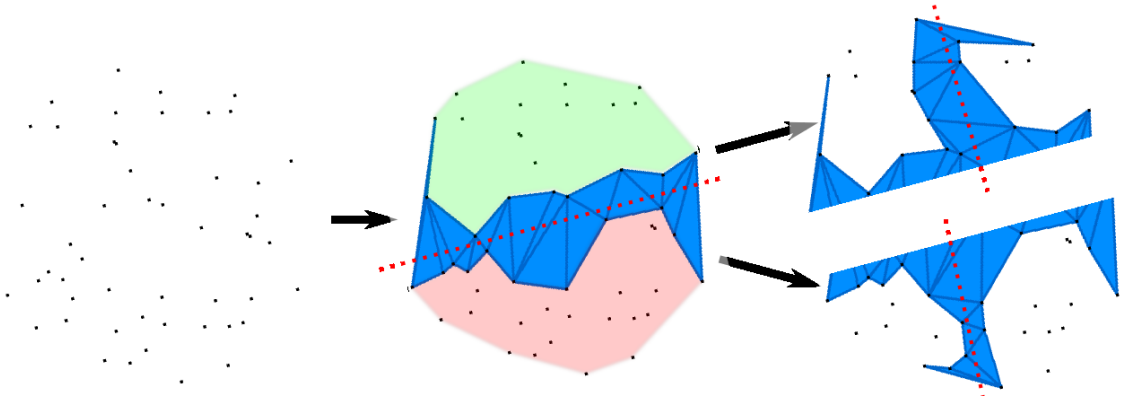

Figura 1: construcción de la primer pared de elementos, que divide el espacio en dos subespacios independientes (verde y rojo), y construcción de las segundas paredes (para cada subespacios), que dividen el problema en cuatro subespacios también independientes

En la implementación propuesta, las paredes de elementos se construyen siguiendo rectas. Para determinar la recta en un trabajo se construye el bounding-box del conjunto de puntos y se toma la recta que pasa por el centro del mismo y es perpendicular a su lado más largo. Para construir los elementos sobre la recta se utiliza un método incremental (McLain, 1976). Este método se basa en buscar, dada una arista de la malla, el nodo (de entre los nodos ubicados en el semiplano hacia el cual avanza la triangulación) cuya distancia desde el centro de la circunferencia que forma con la arista, a la arista misma (distancia con signo, calculada a través del producto vectorial) sea mínima. Si el resultado no es único (por ejemplo, 4 puntos cocirculares), se utiliza un ordenamiento angular respecto de la arista, y siempre en un mismo sentido (por ejemplo en sentido horario) para definir el orden en que se procesan los nodos de igual mínima distancia.

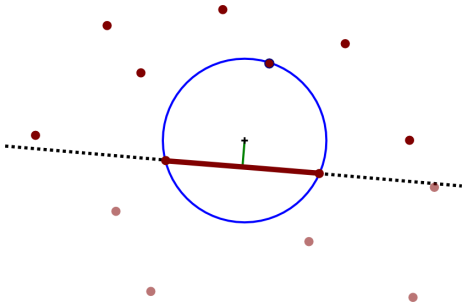

Figura 2: Determinación del nodo adecuado para construir el próximo triángulo. Los nodos inválidos se presentan semitransparentes. Para el nodo seleccionado se muestra el la circunferencia y su centro en relación a la arista

Planteada de esta forma, la búsqueda implica procesar todos los nodos de la malla por cada triángulo que se construye (verificar en cual semiplano se encuentran y, si son potenciales candidatos, construir la circunferencia y hallar la distancia perpendicular a la arista), lo cual resulta

excesivamente costoso en comparación con los métodos utilizados en algoritmo secuenciales. Para lograr tiempos de ejecución más competitivos, en Novara y Calvo (2011) se describen algunas optimizaciones posibles.

Primeramente se utiliza una estructura de celdas para el ordenamiento y la búsqueda espacial para evaluar al avanzar sobre una arista los nodos candidatos a formar un elemento con la misma de forma ordenada. Si en un momento del recorrido se tiene un nodo en particular (in) como nodo más cercano hasta el momento, y la circunferencia que definía este nodo junto con los de la arista tiene radio r, se puede demostrar que cualquier nodo cuya distancia sea mayor a $2r$ no puede generar una circunferencia con centro más cercano que la generada por el nodo in. Utilizando esta propiedad se pueden descartar celdas completas de la estructura sin analizar sus nodos ni calcular las circunferencias. Para que el descarte sea temprano, conviene comenzar la búsqueda por los nodos más cercanos a la arista. La estructura propuesta en el trabajo original es una grilla regular. Esta estructura permite localizar la celda en la que se encuentra un punto en tiempo constante, y permite además recorrer las demás celdas en orden de distancia creciente según la métrica de Chebyshev fácilmente. En la gran mayoría de los casos, no es necesario recorrer más de nueve celdas para determinar el triángulo que cumple la propiedad Delaunay. De esta forma, el algoritmo que originalmente presentaba un orden cuadrático se convierte en un algoritmo con tiempo esperado lineal. La cantidad de celdas utilizadas se determina en forma aproximada de acuerdo a la densidad de puntos, con una función ajustada experimentalmente.

2.1. Frontera Impuesta

El algoritmo descrito en (McLain, 1976) y utilizado en el paso base del algoritmo en paralelo no considera una frontera impuesta. En el caso del mallado en paralelo, solo el primer trabajo tiene libres todas sus fronteras, dado que los demás trabajos deben respetar los elementos generados por los trabajos anteriores. Sin embargo, como se mencionó anteriormente, si todos los elementos cumplen con la condición Delaunay y los casos de ambigüedad se resuelven con un criterio global, estas restricciones de frontera se satisfacen automáticamente, por lo que no es necesario considerarlas. Cuando el algoritmo debe mallar sobre una frontera impuesta cuyas aristas no coinciden con las aristas de la triangulación Delaunay del conjunto de puntos, o cuando a causa del error numérico las situaciones de ambigüedad no se detectan de igual forma en distintos trabajos, se deben realizar nuevas consideraciones para garantizar la validez del resultado. En la implementación mencionada, siempre se toma como dato de entrada una frontera impuesta, que puede o no ser la frontera del convex-hull. Dado que en el caso 2D la pared de elementos a construir avanza sobre una línea recta, se pueden ordenar los segmentos de la frontera que son atravesados por esa recta. De esta forma, la construcción de los elementos puede realizarse en orden, sabiendo en cada momento desde qué arista de la frontera (arista de entrada) se comienza a colocar una secuencia de triángulos y en qué arista debe finalizar (arista de salida), aún cuando el dominio contenga huecos. Antes de colocar cada triángulo, se verifica si las nuevas aristas que el triángulo genera son compatibles con la arista de salida. En caso de ser incompatibles, se genera un último elemento en esa serie que no cumple con el criterio Delaunay, pero que garantiza la validez de la malla resultante. En la figura 3 se marca en lineas de puntos un elemento que habría sido colocado en lugar del elemento 4 de no haber existido la arista B como frontera. El problema se detecta cuando hay intersección entre las aristas del potencial elemento y una arista de salida.

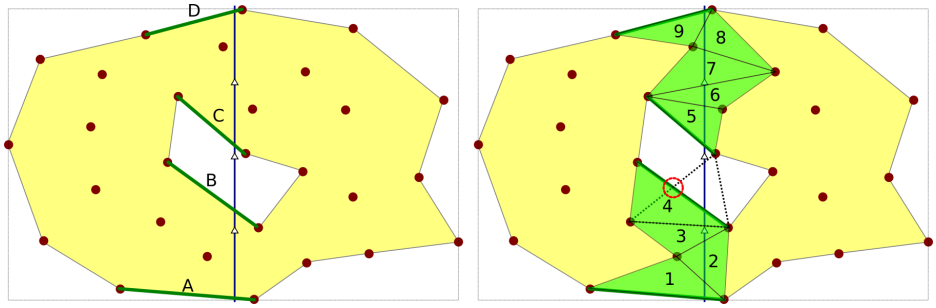

Figura 3: Izquierda: ordenamiento de las aristas de entrada (A,C) y salida (B,D). Derecha: Construcción de las secuencias de triángulos en orden (1 a 4, y 5 a 9). La arista B viola la condición Delaunay.

3. EXTENSIÓN A 3D

La extensión a 3D del algoritmo sin considerar una frontera impuesta y con los nodos en posición general resulta directa. Las paredes de elementos se construyen siguiendo planos medios en lugar de rectas, determinados con el mismo criterio: para un trabajo dado es el plano que pasa por el centro del bounding-box de sus puntos y es perpendicular al lado más largo del mismo. Para construir un elemento a partir de una cara, se busca el centro de esfera con menor distancia con signo a esa cara. En esta sección se presentan algunos aspectos de interés de una implementación del método 3D para arquitecturas de memoria compartida, así como también los resultados obtenidos con diferentes configuraciones del algoritmo y diferentes tamaños de malla.

3.1. Ordenamiento Espacial y Búsqueda

Al igual que en la implementación 2D, se utilizó primero una grilla regular como estructura de ordenamiento espacial, con similares resultados. Las figuras 4 y 5 muestra los tiempos medidos para mallas de 50.000 y 100.000 nodos variando la densidad de la grilla. Se observa que los mejores resultados se obtienen cuando la relación entre la cantidad de nodos y la cantidad de celdas de la grilla está entre 1:1 y 2:1.

nodos/celda	tiempo (s)
802.5	40.53
237.8	22.91
100.3	14.07
51.3	10.17
29.7	7.54
18.7	6.18
12.5	5.13
8.8	4.65
6.4	4.28
4.8	4.00
3.7	3.78
2.9	3.71
2.3	3.60
1.9	3.58
1.6	3.55
1.3	3.55
1.1	3.56
0.9	3.57
0.8	3.59

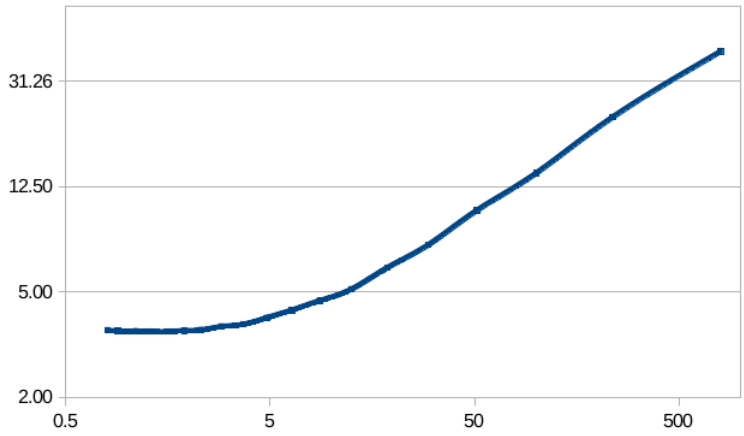

Figura 4: Tiempo de mallado vs densidad de la grilla regular (promedio de nodos por celda) para una malla de 50.000 nodos.

nodos/celda	tiempo(s)
1596.1	154.78
199.5	51.87
59.1	25.78
24.9	15.75
12.8	11.57
7.4	9.37
4.6	8.29
3.1	7.72
2.2	7.40
1.6	7.21
1.2	7.22
0.9	7.22
0.7	7.46
0.6	7.47
0.5	7.62

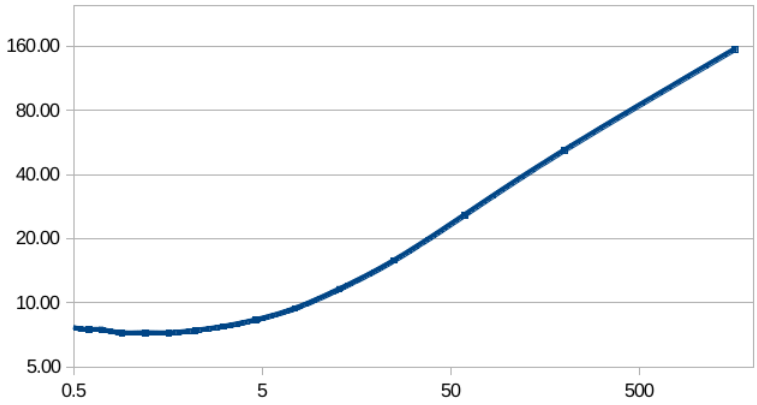

Figura 5: Tiempo de mallado vs densidad de la grilla regular (promedio de nodos por celda) para una malla de 100.000 nodos.

Como alternativa se propuso la utilización de un octree, implementándose también un bucket-octree como variante. El primero es un árbol donde cada hoja contiene a lo sumo un único nodo, mientras que en el segundo las hojas contienen conjuntos de hasta N nodos, siendo N un parámetro a determinar. Un bucket-octree tendrá una profundidad menor que un octree convencional. Sin embargo, dado que al buscar el siguiente nodo para construir un elemento, el descarte se realiza por celda y no por nodo, puede obligar a testear todos los nodos de una celda en casos donde con un octree convencional la mayoría de ellos podrían ser descartados luego de testeados los primeros. Las figuras 6 y 7 muestran los tiempos medidos para para diferentes cantidades de nodos por celda en el octree, para mallas de 50.000 y 100.000 nodos respectivamente. Los resultados muestran que el octree convencional (1 nodo por celda en la gráfica) permite obtener mejores resultados que un bucket-octree. Sin embargo, estos resultados se acercan pero no superan a los que se obtienen con grillas regulares de entre 1 y 2 nodos por celda.

nodos/celda	tiempo
1	3.891
2	8.224
4	14.6
8	11.498
16	10.361
32	10.166
64	9.774
128	9.674
256	15.207
512	18.116
1024	19.899

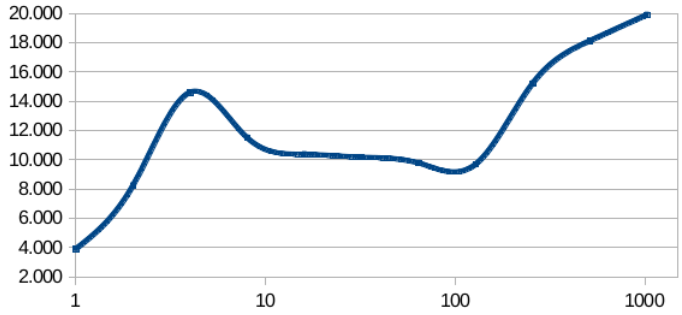

Figura 6: Tiempo de mallado vs cantidad máxima de nodos por celda en un bucket-octree, para una malla de 50.000 nodos.

La figura 8 compara los tiempos del algoritmo sin utilizar ninguna estructura de ordenamiento espacial, utilizando una grilla regular con 1.5 nodos por celda, y utilizando un octree con 1 nodo por celda. El octree incrementa los tiempos respecto a la grilla regular en un 51 % promediando todas las corridas de más de 5000 nodos. Vale destacar que las mallas utilizadas en estas pruebas fueron generadas insertando nodos en posiciones completamente aleatorias con igual probabilidad para todo el dominio. De esta forma, la malla resultante presenta un tamaño

nodos/celda	segs
1	9.231
2	28.917
4	38.455
8	36.3296
16	28.283
32	26.178
64	26.038
128	28.288
256	29.185
512	50.308
1024	63.461

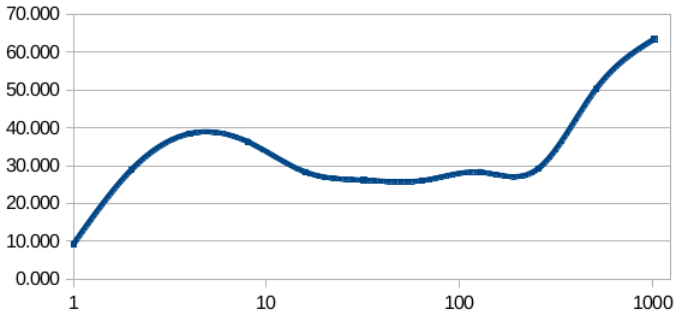

Figura 7: Tiempo de mallado vs cantidad máxima de nodos por celda en un bucket-octree, para una malla de 100.000 nodos.

de elemento (h) aproximadamente constante. Se puede pensar que esta distribución favorece el rendimiento de la grilla, y desaprovecha las ventajas del octree, dado que las únicas variaciones de profundidad en el árbol se deben a la forma del dominio, y no a la densidad de nodos en el mismo. Se realizaron además pruebas similares dividiendo el dominio en 4 sectores de h aproximadamente constante cada uno, pero con tamaños de elemento promedio relativos de h, $2h$, $4h$ y $8h$. Los resultados fueron similares, el octree resultó en promedio 54 % más lento que la grilla regular.

# nodos	ninguna	grilla	octree
1000	0.05	0.05	0.06
2500	0.18	0.12	0.18
5000	0.51	0.25	0.36
10000	1.50	0.52	0.76
25000	6.40	1.36	2.13
50000	19.92	2.76	3.89
100000	62.31	5.73	9.23
250000	315.90	14.57	21.46
500000		29.98	44.00
1000000		60.72	103.31
2500000		159.11	217.28

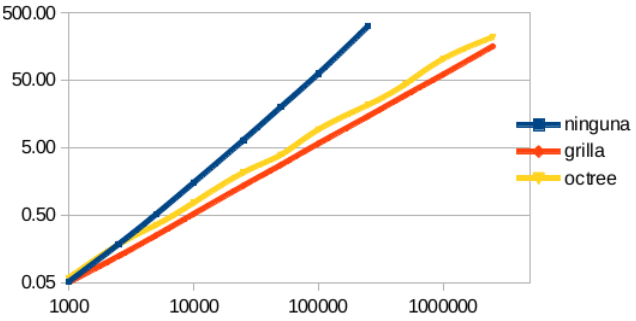

Figura 8: Tiempos de mallado para las diferentes estructuras de ordenamiento y búsqueda espacial propuestas.

3.2. Frontera Impuesta

Dado que la construcción de la pared que divide el dominio avanza sobre un plano, no se puede establecer un orden trivial como se lo hace en 2D. En 3D, cualquier cara de la frontera que sea atravesada por el plano puede considerarse de entrada o de salida, según la situación. El algoritmo que construye la pared de elementos en un trabajo utiliza una pila de caras de entrada que inicialmente contiene las caras de la frontera impuesta, y a medida que avanza estas van siendo reemplazadas por nuevas caras internas. Antes de colocar un elemento nuevo, se debe realizar una verificación entre el potencial elemento y las caras de salida, similar a la utilizada en el caso 2D, y por las mismas razones. Sin embargo, en este caso todas las caras de la frontera son potenciales caras de salida, por lo que verificar cada nuevo elemento contra todas ellas resulta excesivamente costoso. Para solucionar este problema, se modificó la estructura de datos de la malla para que cada nodo contenga referencias a las caras de frontera de las que forma parte. Las caras de frontera a verificar antes de colocar un nuevo elemento serán las caras de los tres nodos

de la cara base sobre la cual se construye el nuevo elemento. Las verificaciones a realizar son similares a las que debe realizar un mallador por avance frontal. Primeramente se analiza que un nuevo elemento no genere caras o aristas que atraviesen a otras caras ya existentes, verificando para cada nodo candidato de qué lado del plano que forman las caras de frontera mencionadas se encuentra. Si se ignoran los problemas producto del error numérico en estas operaciones, esta verificación debería ser suficiente para garantizar la validez del resultado. Sin embargo, debido a que el orden en que se realizan los cálculos necesarios para determinar la validez de un nodo puede variar el resultado, es necesario realizar una verificación adicional para evitar colocar una nueva cara justo sobre una cara existente, en el mismo plano y compartiendo dos vértices, cuyas aristas se intersecten. Si esto se acepta, el espacio nulo entre estas dos caras se debe completar con un sliver extremo (de volumen 0), pero las primeras verificaciones pueden no permitir la inserción de este elemento. Es por esto que cuando se determina un empate entre los puntos que generan esferas con centros de distancia más negativa a la cara de avance, se utiliza aquel cuyas nuevas caras resulten compatibles con las previamente generadas. Además, dada la naturaleza del test que se realiza para cada nodo, un nodo coplanar a la cara de avance generaría una esfera de radio infinito, y por lo tanto la distancia de su centro a la cara de avance sería infinitamente negativa. Según el criterio original, este nodo sería el elegido para avanzar, pero para evitar la formación de slivers, a estos nodos se les asigna una distancia infinitamente positiva, de forma que solo resulten elegidos si ningún otro nodo de la malla supera las verificaciones antes mencionadas. Esto reduce notablemente la generación de slivers, algo muy común en mallas donde los nodos se ubican de forma regular, como en los primeros pasos de una simulación mediante un método de partículas.

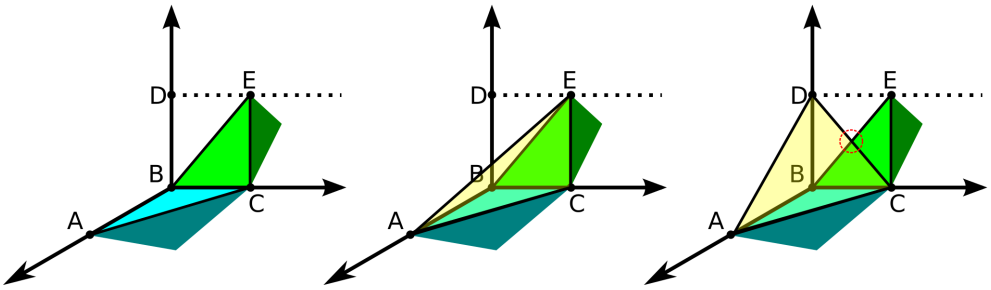

Figura 9: Suponiendo que el mallador avance por la cara ABC, los nodos D y E son igualmente válidos. La presencia de la cara BEC condiciona la elección. En el caso de la derecha, se debería incluir un tetraedro degenerado totalmente plano BDCE para conectar las caras resultantes. El algoritmo detecta la intersección señalada en rojo y evita esta situación cuando es posible.

4. RESULTADOS, CONCLUSIONES Y TRABAJOS FUTUROS

En la figura 10 se muestran los tiempos medidos para mallas de h constante de 50.000, 250.000 y 1.000.000 de nodos utilizando tanto la grilla regular como un octree, variando el número de hilos de ejecución en paralelo. Para las pruebas se utilizó una PC con 4 núcleos reales, pero con tecnología HyperThreading, razón por la cual se incluyen en la tabla corridas con 8 hilos en paralelo, pero no se consideran en el análisis de resultados. Se observa que la eficiencia paralela, si bien disminuye conforme aumenta el número de hilos, se mantiene sobre el 70 % para el hardware de prueba cuando se utiliza una grilla regular, y presenta un rendimiento levemente inferior cuando se utiliza un octree. Se observa también que la eficiencia aumenta cuando aumenta el número de nodos, debido a que la mayor cantidad de trabajos a

realizar en total para el mallado aumenta, reduciendo el peso relativo de las primeras etapas del algoritmo donde solo algunos procesadores realizan trabajo.

# nodos	50000				250000				1000000			
	grilla		octree		grilla		octree		grilla		octree	
# threads	tiempo	ef. par.	tiempo	ef. par.	tiempo	ef. par.	tiempo	ef. par.	tiempo	ef. par.	tiempo	ef. par.
1	2.83		4.06		14.86		22.74		61.75		110.75	
2	1.64	86.36	2.38	85.25	8.38	88.67	13.27	85.66	34.88	88.52	63.96	86.58
3	1.26	75.19	1.86	72.84	6.27	79.06	10.27	73.77	25.61	80.37	47.66	77.45
4	1.04	67.99	1.63	62.17	5.06	73.41	8.59	66.16	20.53	75.20	39.03	70.93
8	1.00	35.46	1.51	33.53	4.61	40.30	8.23	34.54	18.35	42.07	35.26	39.26

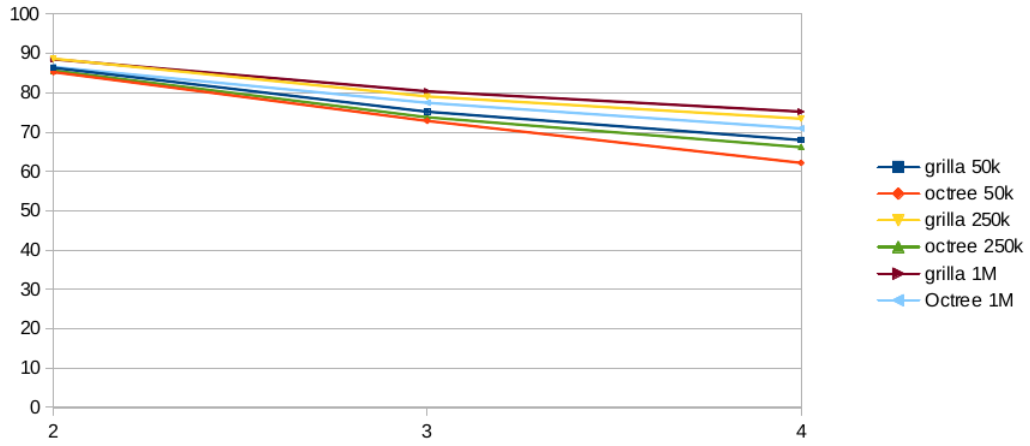

Figura 10: Eficiencia paralela en función del número de hilos de ejecución

4.1. Memoria compartida

Como trabajos futuros, se deben seguir evaluando diferentes mejoras con el objeto aumentar la eficiencia paralela en la implementación para arquitecturas de memoria local presentada. Hay dos aspectos centrales en esta implementación. Por un lado las estructuras de búsqueda y ordenamiento espacial son estructuras convencionales; resta analizar el funcionamiento de variaciones de estas estructuras o tal vez plantear estructuras alternativas ad-hoc para este problema. Por otro lado, los pasos iniciales del proceso, en los que no todos los procesadores tienen tareas asignadas, son los pasos con mayor número de nodos a procesar, y por ende los que más tiempo consumen. Se deben implementar y evaluar estrategias de paralelización específicas para aprovechar esa capacidad de cálculo extra en estos primeros pasos, buscando que varios hilos resuelvan en conjunto un mismo trabajo. En este sentido, aún queda un margen importante de mejora para la eficiencia paralela en este tipo de arquitecturas.

4.2. Memoria local

Se han realizado pruebas en arquitecturas de memoria local con la implementación 2D del algoritmo. Los resultados no han sido satisfactorios para ese caso, ya que se observó que el overhead generado por la comunicación de los puntos de la malla entre los nodos del cluster superaba ampliamente la ganancia en los tiempos de cómputo producto de la división del trabajo entre procesadores. En el caso 3D, la relación entre el volumen de cálculos a realizar y la cantidad de información a transmitir es más favorable comparativamente, aunque no se han realizado pruebas en este sentido para determinar su verdadero impacto. Sin embargo, en una aplicación real el software de mallado se debe integrar en un software más complejo, como por

ejemplo un solver P-FEM. Dado que no es necesario que el algoritmo de mallado divida los trabajos de forma balanceada, sino que un mal balance inicial puede corregirse sobre la marcha, se puede intentar re-aprovechar la distribución de nodos del solver para reducir los tiempos de comunicación iniciales y obtener así un mejor aprovechamiento de una arquitectura de memoria local.

REFERENCIAS

Cignoni P., Montani C., y Scopigno R. Dewall: A fast divide and conquer delaunay triangulation algorithm in ed. *Computer-Aided Design*, 30(5):333–341, 1998.

McLain D.H. Two dimensional interpolation from random data. *Comput. J.*, 19(2):178–181, 1976.

Novara P. y Calvo N. Implementación de un método paralelo de triangulación delaunay euclídeo. *XIX Congreso Sobre Métodos Numéricos y sus Aplicaciones (ENIEF)*, 2011.